ITS assignment 3

Noah Jensen (xdr622) & Julian Pedersen (rsk975)

September 2022

SEED

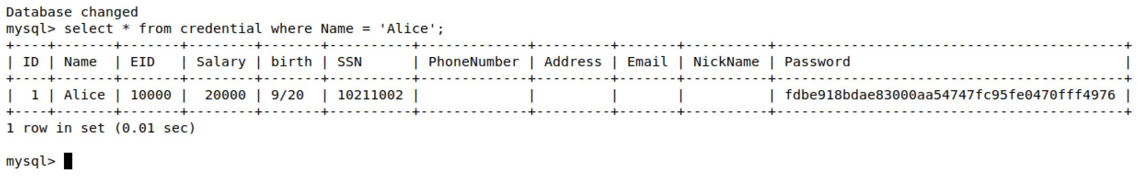
Task 1

In this task we had to get familiar with dockers and mysql. We had to set up a docker and be able to print all information on the employee Alice.
First, we downloaded the provided Labsetup.zip file and unzipped it to a directory. Next, we built a docker container using the `docker-compose build` and `docker-compose up` commands. We added a new entry to the `/etc/hosts/ 10.9.0.5 www.seed-server.com` to (WHY)

Finally, we open a container in the above host with `docksh <id>`.

We run the command `mysql -u root -pdees` to open a SQL session and `use sqllab_users;` to load the supplied database.

We then had to print all known information of the employee named Alice, we did this by the following SQL-statement: `select * from credential where Name = 'Alice';` this resulted in the following screenshot



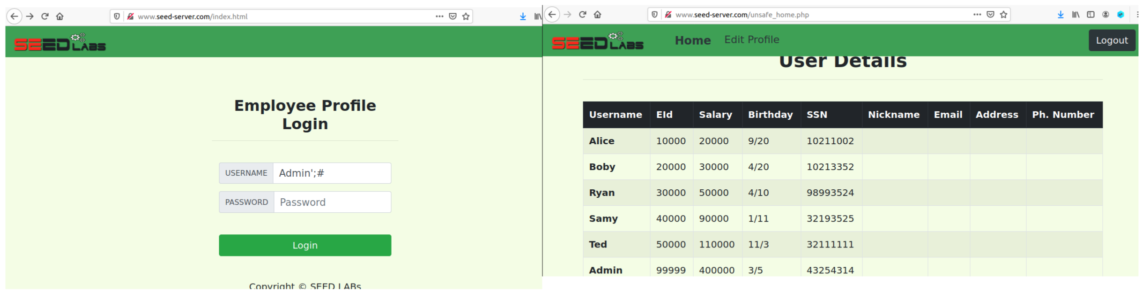
```
Database changed
mysql> select * from credential where Name = 'Alice';
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.01 sec)

mysql> 
```

Figure 1: All information on Alice

Task 2

Task 2.1

To gain access to the admin account we used the username field and wrote `Admin';--`. What this SQL-statement did was comment out the password authentication line by placing the double-dash at the end of the line, whilst providing a semicolon before the comment, such that our SQL-statement would still execute. In this way we skipped the Password authentication step and was able to login in as admin successfully. We found that our SQL injection did not work if we did not include a whitespace after the double dash. After a quick search in the MySQL documentation we learned that a double dash is only recognized as a comment if it is following by at least one whitespace or control character. However when we used the `#` character as the comment style, the whitespace was not required.



Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Figure 2: Our SQL-statement in the browser and after successful login

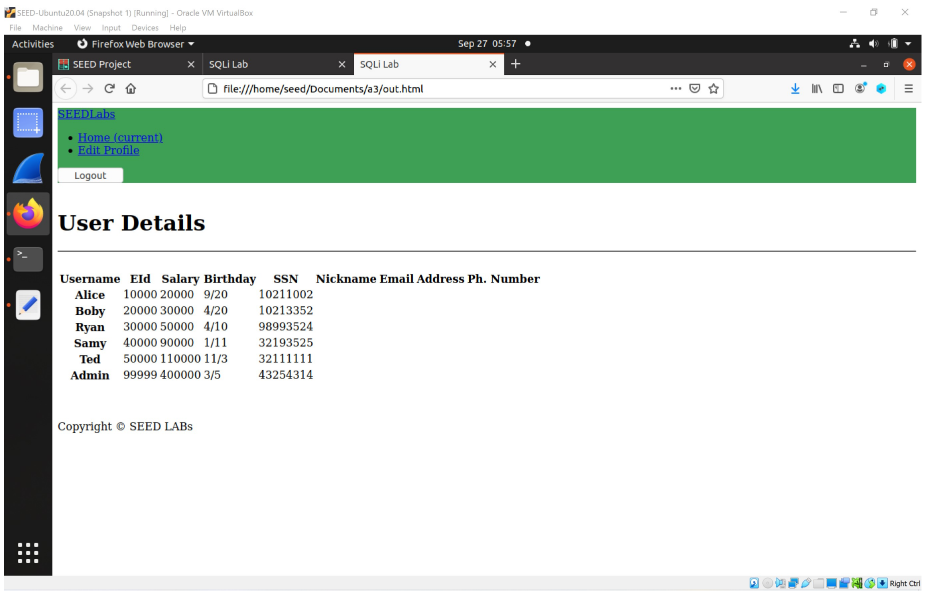
Task 2.2

We had to repeat the exercise but this time in the terminal. We used the following `curl` command

`curl 'www.seed-server.com/unsafe_home.php?username=admin%27%3B%2D%2D%20&Password=11'`

. The special characters had to be written in their hex format for the request to go through.

By piping the output to a new file `out.html`, we can show the data in a more readable format:


SEED-Ubuntu20.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Firefox Web Browser
Sep 27 05:57
SEED Project
SQLi Lab
SQLi Lab
file:///home/seed/Documents/a3/out.html
SEEDLabs
Home (current)
Edit Profile
Logout
Right Ctrl

User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Copyright © SEED LABs

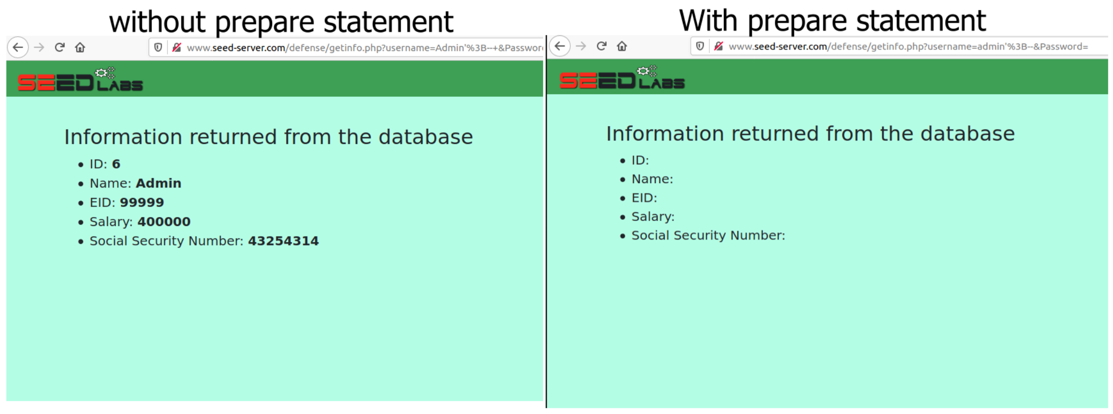
Figure 3: after logging in through the terminal

Task 2.3

We tried to run the following two statements in the login page `Admin’; DELETE FROM credentials WHERE Name=’Alice’;--` . This did not work, we got an error and did not login in to the page nor delete Alice from the database. This is because the query statement used in the websites PHP code does not support multiple queries. To execute multiple SQL queries, each SQL statement must be invoked as a connection query separately (alternatively, they must use the `multiquery() command`. We do not have the power to make the underlying PHP code execute more than one query in an SQL injection attack, so instead we receive an error.

Task 4

In this task we had to make a defense against a SQL injection attack. We this buy editing in the provided unsafe.php (see appendix for changes). We implemented a prepare statement to help sanitize user input. Before the prepare statement we could get info about admin by using the SQL injection `Admin’;--` . But after we made the prepare statement our attack did not grant us any information. This is seen in figure 4.



www.seed-server.com/defense/getinfo.php?username=Admin'%3B--+&Passwor

Information returned from the database

- ID: **6**
- Name: **Admin**
- EID: **99999**
- Salary: **400000**
- Social Security Number: **43254314**

www.seed-server.com/defense/getinfo.php?username=admin'%3B--&Password=

Information returned from the database

- ID:
- Name:
- EID:
- Salary:
- Social Security Number:

Figure 4: Before and after prepare statement with the SQL injection `Admin’;--`

The SQL execution process nominally goes through 6 steps: parsing, semantics checking, binding, query optimisation, caching, and execution. A prepared statement is similar, but the binding step differs by compiling the code with placeholders instead of actual data. The input data is then supplied in the step between caching and execution. In this way, user input is not compiled along with the rest of the code, but instead treated as raw input to specially prepared placeholders, thus ensuring that even if the input could be valid code to execute, it never will be executed, as it is held separate from the compilation process.

Short answers

Question 1

Define a buffer overflow attack.
a buffer overflow attack is an attack where you overwrite parts of the memory you should not have access to. It is possible to exploit this vulnerability by overwriting memory, forcing the machine to execute malicious code – e.g. overwriting a procedure return address to point to system code (stack-based return-to-libc attack) . Buffer overflow attacks can also reveal part of the memory that the user should not be able to see such as keys.

Question 2

List and briefly describe one or more defences against stack-based buffer overflows.
One defence could be making a check that sanitizes user-input such that the user can not force to program to write to more memory than specifically allocated to the write buffer. Another defence could be implementing flags for non-executable memory. Another defense is run-time stack protection. A *stack canary* – a checkword used to detect code injection – can be inserted into an extra field just below attack targets (e.g. return addresses, local variables). Checking whether such code has been overwritten at runtime allows the program to interrupt execution to an error handler instead of executing the malicious code.

Question 3

If a website only allows `HTTP` connections, is it more or less at risk from `SQL` injection attacks?
Equally at risk, because `SQL` come from user-inputs while the `S` in `HTTPS` stand for `SSL` which provides a secure connection. Thus the `SSL` does not stop `SQL` injection attacks because it does not input validate. It is a input validation problem and not a user authentication problem.

Question 4

Suppose you find a vulnerability in the 'unzip' command on Unix that you can exploit to get a shell. You share your finding with your study companion who says that because the unzip program is owned by root, you can SSH into ssh.diku.dk and run your exploit to get a root shell. You disagree, arguing that running the command would give you a shell running with your privileges, not root privileges. Who is right, and why?
By default, SSH is configured to deny root access to users. Even though the

unzip command is owned by root, all groups have execution and reading rights to the unzip command. Thus the result of the exploit would give us a shell running with our user privileges not root. However, if the exploit includes changing the Set-UID bit, then running this exploit would give root access.

Appendix

Task 4 changed code

```
  <?php
// Function to create a sql connection.
function getDB() {
  $dbhost="10.9.0.6";
  $dbuser="seed";
  $dbpass="dees";
  $dbname="sqllab_users";

  // Create a DB connection
  $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
  if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error . "\n");
  }
  return $conn;
}

$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);

// create a connection
$conn = getDB();

$stmt = $conn->prepare("SELECT id, name, eid, salary, ssn
            FROM credential
            WHERE name = ? and password = ? ");

// Bind params to query
$stmt->bind_param("ss", $name, $password);
$stmt->execute();
$stmt->bind_result($bind_id, $bind_eid, $bind_salary, $bind_ssn);
$stmt->fetch();

/*
// do the query
 $result = $conn->query("SELECT id, name, eid, salary, ssn
                    FROM credential
                    WHERE name= '$input_uname' and Password= '
   $hashed_pwd'");
if ($result->num_rows > 0) {
  // only take the first row
  $firstrow = $result->fetch_assoc();
  $id     = $firstrow["id"];
  $name   = $firstrow["name"];
  $eid    = $firstrow["eid"];
  $salary = $firstrow["salary"];
  $ssn    = $firstrow["ssn"];
*/
}

// close the sql connection
$conn->close();
?>
```